Phaint

Distributed System project

Farinelli Ettore (farinelli.ettore@studio.unibo.it)

Ghignatti Nicolò (ghignatti.nicolo@studio.unibo.it)

Academic Year: 2024/2025

Abstract

This project proposes the development of a **distributed drawing and mock-up tool** that combines the simplicity of Paint with the collaborative features of Figma. The platform will enable users to create mock-ups through freehand drawing while supporting real-time, multi-user collaboration. By introducing a distributed architecture, mock-ups will be seamlessly shared across users with configurable access rights, ensuring concurrent editing and version consistency.

Phaint aims to empower teams with an intuitive yet powerful environment for ideation, sketching, and prototyping, leveraging both the accessibility of simple drawing tools and the "superpowers" of collaborative design systems.

Goals

The project aims to set high standards for usability and reliability while innovating within its problem domain. Additional objectives included adhering to project timelines, maintaining transparency in development, and producing a maintainable codebase that supports future enhancements.

Achievements

Phaint's key achievements include the successful achievement of the specified goals and the creation of a feature-rich application.

1. Concept

1.1. Type of Product

Phaint is a **web-based collaborative application** with a graphical user interface, designed for use through desktop browsers and potentially optimized for mobile access. The system supports real-time drawing and collaboration, allowing multiple users to interact on shared canvases to create, edit, and share mock-ups or visual diagrams in a distributed environment.

1.2. Use Case Collection

- **Where are the users?** Users can access the application from any location, as long as they have an internet connection and a modern web browser. This includes remote teams, distributed workgroups, and individual users who work from home, office, or public spaces.
- **When and how frequently do they interact?** Interaction with the system is expected to vary: some users may log in occasionally to review or contribute to a project, while others may engage in daily or real-time collaboration sessions depending on project needs or creative work schedules.

- **How do they interact? What devices are they using?** Interaction is primarily through a web browser, using either desktop computers or mobile devices. The user interface provides intuitive drawing and collaboration tools accessible with mouse, touch screen, or stylus input.
- **Does the system store user data? Which? Where?** The application securely stores user registration details (email, username, password) and project-related information such as project names, drawing data, and collaborative history. Data is managed via a backend database or cloud storage, ensuring accessibility and data integrity.
- **Are there multiple roles?** The system will have to distinguish between project owner and collaborator, but no particular external roles will be needed.

2. Requirements

2.1. Functional Requirements

- **Registration/Login:** The system must allow users to register and log in.
  Acceptance criteria: A new user can create an account and log in securely.
- **Mock-up Creation and Editing:** The system must allow users to create mock-ups using freehand drawing and shape tools.
  Acceptance criteria: A user can open a blank canvas, draw and add components, and save the work.
- **Real-Time Collaboration:** Multiple users must be able to edit the same mock-up simultaneously, with updates visible instantly.
  Acceptance criteria: Edits made by one user appear on all collaborating users' screens without delay or conflict.

2.2. Non-Functional Requirements

- **Availability:** The system must be available always.
  Acceptance criteria: The system must have at least 99.5% uptime.
- **Consistency:** Concurrent edits must remain synchronized and conflict-free.
  Acceptance criteria: No overlapping or disappearing elements occur during concurrent editing.
- **Usability:** The application must provide an intuitive interface.
  Acceptance criteria: Usability tests show that 80% of new users can complete basic tasks without assistance.
- **Scalability:** The system must handle increasing numbers of concurrent users without performance degradation.
  Acceptance criteria: Load testing demonstrates support for large numbers of users.

2.3. Implementation Requirements

- **Technology stack:** The application must be developed as a web application using JavaScript/TypeScript frameworks for the frontend and frameworks supporting RESTful APIs and WebSocket for the backend.
- **Data Storage:** All user and project data must be stored in a secure, persistent database.

2.4. Optional Requirements

- **PDF export:** There should be a possibility for users to export their work; PDF could be a good choice because it allows insertion of annotations and effects.
- **Drawing enhancements:** Allow users to enhance drawing with actions to create better mockups.

3. Design

The system adopts a **client-server architectural style**. The backend employs a RESTful API for standard resource operations and a **WebSocket-based subsystem** for real-time collaboration—the dual approach maximizing both scalability and responsiveness.

Real-time communication uses protocols such as WebSockets, allowing bi-directional, low-latency updates. This ensures all users receive changes instantly with conflict resolution handled centrally.

3.1. Infrastructure

Key infrastructure components:

- **Web Clients:** Run in user browsers on any device and connect securely to the backend.
- **Web Server:** Implements both REST API and WebSocket handler for multiplexed sessions.
- **WebSocket Collaboration Hub:** Each active project has a dedicated hub managing client sessions and synchronizing vector drawing operations using channels and mutex-protected state.
- **Authentication Service:** Managed via Firebase Authentication, decoupled from core logic.
- **Database:** Cloud-based Firestore stores durable data (users, projects, canvases, invitations, roles).

Communication uses secure HTTPS and WSS protocols, with DNS addressing and load balancing by host provider.

3.2. Modelling

Key domain entities:

- **User:** Account info including credentials and roles.
- **Project:** Metadata, collaborator list, canvas data.
- **Canvas/Mock-up:** Stores vector graphic elements and history.
- **Invitation:** Encodes permissions and joinable links.

3.3. Backend Component Relationships

- **User Client → HTTP Server (REST):** Authentication, CRUD on projects and invitations.
- **User Client ↔ HTTP Server (WebSocket /connect):** Real-time synchronization of canvas state.
- **Server → Firebase (Firestore):** Persistent storage and retrieval.

3.4. Types of Messages Exchanged

WebSocket messages include:

- **Commands:** User drawing actions (strokes, shape additions).
- **Events:** Notifications for synchronizing drawing state and cursor positions.
- **Queries:** Initial connection state requests.
- **State Updates:** Full or partial canvas data synchronization.

All messages are JSON structured with type/subtype and payloads.

4. Interactions

Clients send REST calls for operations (login, project CRUD) and maintain persistent WebSocket connections for real-time editing.

The collaboration hub broadcasts commands as JSON events, updating in-memory canvas state and synchronizing clients.

Clients query current state on session join and receive instant updates on collaborative actions.

4.1. State of the System

State data includes:

- User IDs, usernames, live presence info (cursor, color).
- Project metadata including collaborators and access roles.
- Canvas data: geometric vector shapes (paths, circles, rectangles) with style and behavior.
- Invitation links and permissions.
- Session in-memory state: connected users and ongoing drawings.

Persistence is cloud-based with some cached in-memory for sessions.

4.2. Data and Consistency Issues

Data stored:

- Users: auth tokens, preferences.
- Projects: metadata, collaborators, history.
- Canvases: vector elements and actions.
- Invitations: secure tokens.

Firestore's document-oriented model supports hierarchy and efficient queries.

4.3. Fault Tolerance

- Cloud database replication for data durability.
- WebSocket sessions use heartbeat and timeout to detect failures.
- Automatic retries and partial failure routines to maintain partial functionality.
- Graceful degradations maintain read access if real-time inputs fail.

5. Implementation

5.1. Network Protocols

The system uses a combination of HTTP/HTTPS for RESTful resource access (e.g., user registration, project management, invitation handling) and WebSockets for real-time, bidirectional communication required by the collaborative drawing feature.

Listing 1: RESTful API server in Go

```
func main() {
    err := services.FirebaseDb().Connect()
    if err != nil {
        log.Println(err)
    }
    log.Println("Creating the server on port 8080")
    mux := http.NewServeMux()
    // adding all the handlers
    mux.Handle("/users", &handlers.UserHandler{})
    mux.Handle("/projects", &handlers.ProjectHandler{})
    mux.Handle("/invitations/accept", &handlers.InvitationHandler{})
    mux.Handle("/invitations", &handlers.InvitationHandler{})

    // Add WebSocket handler
    mux.Handle("/connect", &handlers.WebSocketHandler{})

    // Run the server
    err = http.ListenAndServe(":8080", mux)
    if err != nil {
        log.Panic(err)
    }
}

// Example project handler
func (p *ProjectHandler) ServeHTTP(w http.ResponseWriter, r *http.
    ↪ Request) {
    switch {
    case r.Method == http.MethodGet:
        p.getProjects(w, r)
        return
    case r.Method == http.MethodPost:
        p.addProject(w, r)
        return
    }
}
```

5.2. Hub Structure

We decided to represent the projects through an hub structure, each user joining the drawing session in the project will be connected with the project Hub, which maintains a list of all the connected clients.

```
type Hub struct {
    clients        map[*Client]bool
    broadcast      chan []byte
    register       chan *Client
    unregister     chan *Client
    users          map[string]*UserPresence
    mutex          sync.RWMutex
    projectID      string
    workBoard      *services.CanvasService
    projectHandler *ProjectHandler
}

type Client struct {
```

```
    hub      *Hub
    conn     *websocket.Conn
    send     chan []byte
    userID   string
    username string
}
```

5.3. In-Transit Data Representation

All data exchanged between clients and server, including REST requests/responses and WebSocket messages, is serialized as **JSON**. JSON offers a lightweight, human-readable, and widely supported format

The data are represented through different structs:

```
type VectorShape struct {
    ID          string  `firestore:"id" json:"id"`
    Stroke      string  `firestore:"stroke" json:"stroke"`
    StrokeWidth float64 `firestore:"strokeWidth" json:"strokeWidth"`
    Fill        string  `firestore:"fill" json:"fill"`
    Action      Action  `firestore:"action" json:"action"`
}

type VectorPath struct {
    VectorShape
    Type   string  `firestore:"type" json:"type"`
    Points []Point `firestore:"points" json:"points"`
}

type VectorRectangle struct {
    VectorShape
    Type   string  `firestore:"type" json:"type"`
    X      float64 `firestore:"x" json:"x"`
    Y      float64 `firestore:"y" json:"y"`
    Width  float64 `firestore:"width" json:"width"`
    Height float64 `firestore:"height" json:"height"`
}

type VectorCircle struct {
    VectorShape
    Type   string  `firestore:"type" json:"type"`
    CX     float64 `firestore:"cx" json:"cx"`
    CY     float64 `firestore:"cy" json:"cy"`
    Radius float64 `firestore:"radius" json:"radius"`
}

type VectorElement interface{}

type VectorData struct {
    Width          float64         `firestore:"width" json:"width"`
    Height         float64         `firestore:"height" json:"height"`
    BackgroundFill string          `firestore:"backgroundFill" json:"
        ↪ backgroundFill"`
    Elements       []VectorElement `firestore:"elements" json:"
        ↪ elements"`
    Timestamp      string          `firestore:"timestamp" json:"
        ↪ timestamp"`
```

```
    Version         string          `firestore:"version" json:"version"
        ↪ `
}

type Canvas struct {
    ID          string     `firestore:"id" json:"id"`
    VectorData VectorData `firestore:"vectorData" json:"vectorData"`
}
```

The idea of this kind of data representation allows the use of vector representation. Unlike raster images made of pixels, vector graphics use geometric primitives such as points, paths, rectangles, and circles that are independent of resolution. This brings several key advantages to real-time collaboration:

- **Scalability**: Vector data can be scaled up or down without losing quality, crucial for diverse device displays and zoom levels.
- **Editability**: Individual drawing elements can be easily modified, styled, or animated without redrawing the entire canvas.
- **Compactness**: Vector representations are often more compact than pixel data, reducing bandwidth and storage needs during collaboration.
- **Semantic Richness**: Structured vector shapes allow attaching interactive metadata (e.g., behaviors, links) to components, enhancing interactivity.

Vector graphics take a completely different approach from raster images storage—**they store mathematical descriptions** of shapes rather than pixel data. Instead of recording "pixel at coordinate (100,200) is blue," vectors store instructions like "draw a circle with center at (100,200) and radius 50 pixels". The storage savings are dramatic, and thanks to this kind of representation, the size of messages exchanged between collaborators and server are lightweight, optimizing performance.

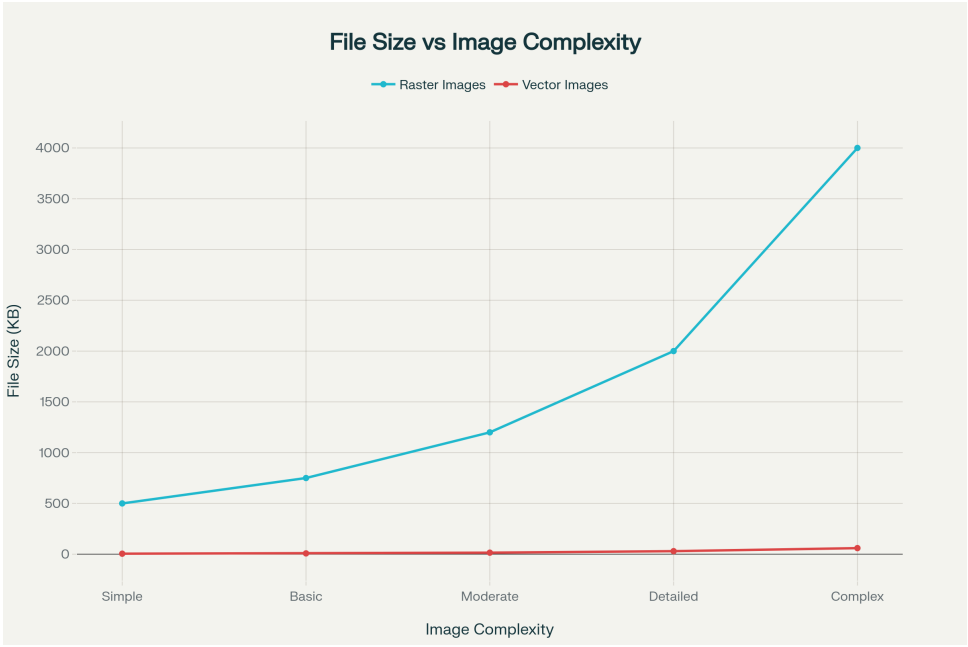

Figure 1: Comparison between memory consumption

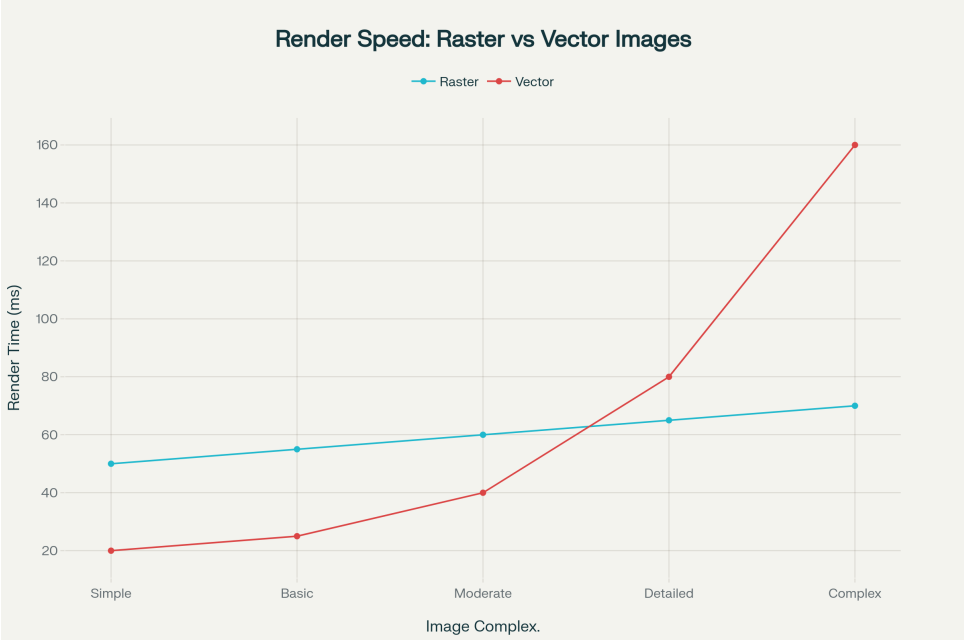

Figure 2: Comparison between render time

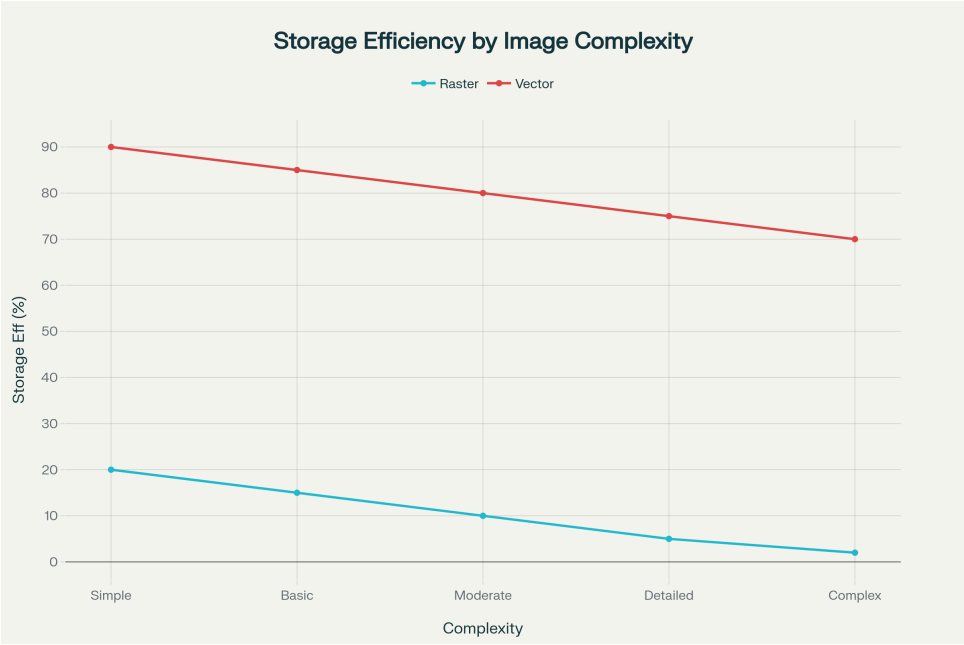

Figure 3: Comparison of storage efficiency

5.4. Database Access and Queries

Backend uses Google Firebase Firestore (NoSQL document database) for persistent data storage.

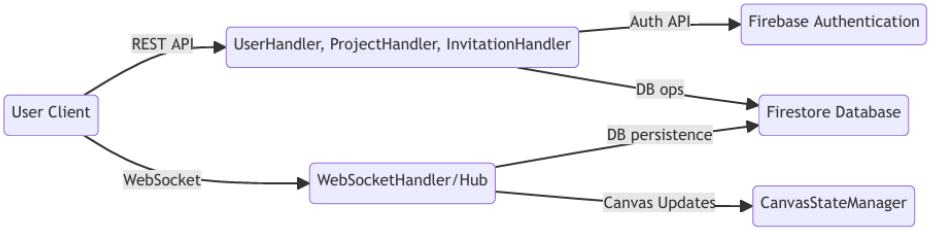

Figure 4: Type of access to the NoSQL db

5.5. Authentication

User authentication is delegated to **Firebase Authentication**, which manages secure user signup, login, password management, and token issuance.

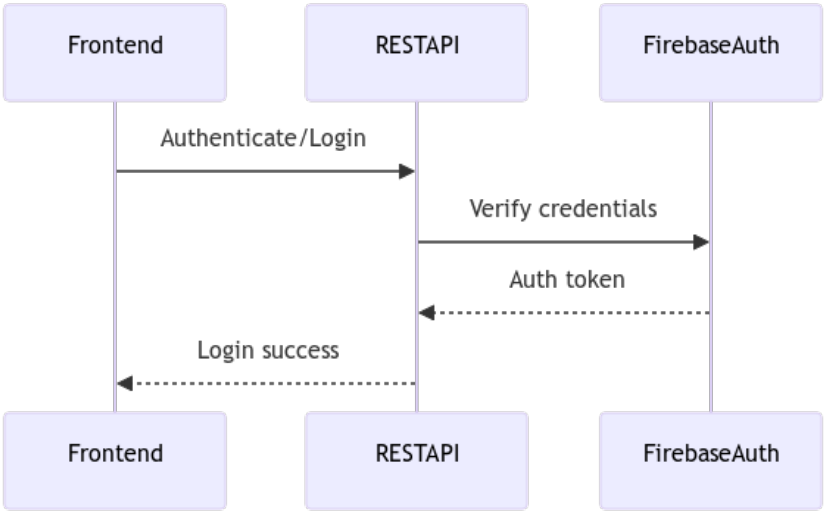

Figure 5: Authentication phases

5.6. Drawing

As said before, Vector Shapes are the backbone of the app's drawing model, which resemble the backend one, in order to simplify the data serialization and deserialization.

```
interface VectorElement {
  id: string;
  type: 'path' | 'rectangle' | 'circle';
  stroke: string;
  strokeWidth: number;
  fill?: string;
  points?: { x: number; y: number }[];
  cx?: number; // circle center x
  cy?: number; // circle center y
  radius?: number;
  x?: number; // rect top-left x
  y?: number; // rect top-left y
  width?: number;
  height?: number;
  action?: { type: string; link: string };
```

```
}
```

Great plus in our implementation is the usage of stores in for the state management, which allow to build reactive applications.

```
import { writable } from 'svelte/store';

function createShapesStore(initialShapes = []) {
  const { subscribe, set, update } = writable(initialShapes);

  return {
    subscribe,
    set,
    addShape: (shape: VectorElement) => update(shapes => [...shapes,
       ↪ shape]),
    removeShape: (id: string) => update(shapes => shapes.filter(s => s
       ↪ .id !== id)),
    updateShape: (updatedShape: VectorElement) =>
      update(shapes => shapes.map(s => s.id === updatedShape.id ?
         ↪ updatedShape : s))
  };
}
```

Communication is handled using a `ProjectWebSocket` class instance with the following behavior:

- Connects using project and user IDs.
- Sends and receives operations representing vector data changes.
- Operations include full canvas data, strokes added or modified, and background fill changes.

Whenever vector data changes locally, the application sends updates:

- For shape changes: sends the updated shape vector.
- For background fill changes: sends the fill value.
- For canvas add/remove: sends respective operations.

Incoming data from the WebSocket is merged into local state stores to synchronize the user interface.

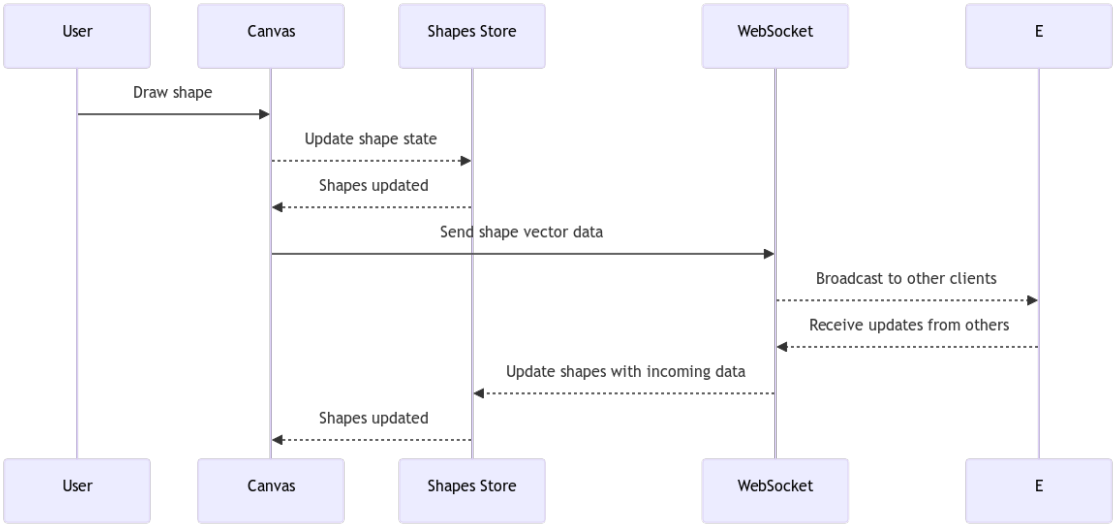

Figure 6: Message exchange during drawings

The flow of this collaboration can be represented as follows:

Figure 7: Drawing flowchart

5.7. Drawings Enhancements

In order to accomplish the challenge to enhance our drawings we decided to divide the action assignment process in two step:

Shape selection uses mouse events to select shapes via marquee or cursor:

```
function handleCanvasMouseDown(event: MouseEvent): void {
  if (get(currentTool) === 'selection') {
    const coords = getCanvasCoordinates(event, canvas);
    selectionMarqueeStart = coords;
    selectionMarqueeEnd = coords;
    isDraggingSelection = false; // Not dragging yet
    selectedShapeIds.set(new Set()); // Clear selection
  } else if (get(currentTool) === 'cursor') {
    const coords = getCanvasCoordinates(event, canvas);
    get(vectorData).elements.forEach(element => {
      if (isPointInVectorElement(coords, element)) {
        dispatch('clickedElement', element);
      }
    });
  }
  // remainder of code omitted for brevity
}

function handleCanvasMouseMove(event: MouseEvent): void {
  sendCursorPosition(getCanvasCoordinates(event, canvas));
  const coords = getCanvasCoordinates(event, canvas);
```

```
  if (get(currentTool) === 'selection') {
    if (!selectionMarqueeStart) return;
    selectionMarqueeEnd = coords;
    redrawCanvas();
    drawSelectionMarquee(selectionMarqueeStart, selectionMarqueeEnd);
  }
  // remainder of code omitted
}

function handleCanvasMouseUp(event: MouseEvent): void {
  if (get(currentTool) === 'selection') {
    if (!isDraggingSelection) {
      const rect = getMarqueeRectangle(selectionMarqueeStart!,
        ↪ selectionMarqueeEnd!);
      selectShapesInRect(rect);
      selectionMarqueeStart = null;
      selectionMarqueeEnd = null;

      const coords = getCanvasCoordinates(event, canvas);
      if (get(selectedShapeIds).size > 0) {
        isDraggingSelection = true;
        dragStartPos = coords;
        captureInitialShapePositions(event);
      }
    } else {
      isDraggingSelection = false;
      dragStartPos = null;
      initialShapePositions.clear();
    }
  }
  // remainder omitted
}
```

Behavior assignment uses a modal allowing users to assign actions to selected shapes:

```
function confirmBehavior(): void {
  if (selectedAction === 'none') {
    showBehaviorModal.set(false);
    return;
  }

  canvases.forEach((canvas) => {
    const shapesArray = get(canvas.shapes);
    shapesArray.forEach((shape) => {
      if (get(selectedShapeIds).has(shape.id)) {
        if (selectedAction.startsWith('goto')) {
          const action = {
            type: 'goto',
            link: gotoPageNumber.toString()
          };
          shape.action = action;
          socket.sendAction(canvas.id, shape.id, action);
        }
      }
    });
    canvas.shapes.set(shapesArray);
  });

  showBehaviorModal.set(false);
```

```
}
```

5.8. PDF Export Functionality

```
export async function downloadAsPDF(canvases: VectorData[]): Promise<
    ↪ void> {
  const pdfDoc = await PDFDocument.create();

  // ... (some code)

  let idx = 0;
  for (const canvas of canvases) {
    const page = pdfDoc.getPage(idx);

    // Fill background color
    const bgColor = hexToRgbNormalized(canvas.backgroundFill);
    page.drawRectangle({
      x: 0,
      y: 0,
      width: canvas.width,
      height: canvas.height,
      color: rgb(bgColor.r, bgColor.g, bgColor.b)
    });

    for (const element of canvas.elements) {
      const stroke = hexToRgbNormalized(element.stroke);
      const fill = hexToRgbNormalized(element.fill);

      // draw element and annotate links here...
    }
  }

  const pdfBytes = await pdfDoc.save();

  // Trigger browser download
  const blob = new Blob([pdfBytes], { type: 'application/pdf' });
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a');
  a.href = url;
  a.download = 'vector_canvases_with_links.pdf';
  a.click();
  URL.revokeObjectURL(url);
}
```

The crucial part was to add the link annotation, cause no library provide an intuitive way to do it, basically when the function is used, it marks out a rectangle (the clickable area) on a chosen page of the PDF, and programs it to transport the reader to a different page when clicked. If the page already has clickable areas, it adds this one to the list; if not, it creates a new list and adds it.

```
function createUriActionDict(url: string, context: PDFContext):
    ↪ PDFDict {
  const dict = PDFDict.withContext(context);
  dict.set(PDFName.of('Type'), PDFName.of('Action'));
  dict.set(PDFName.of('S'), PDFName.of('URI'));
  dict.set(PDFName.of('URI'), PDFString.of(url));
```

```
  return dict;
}

export function addLinkAnnotation(
  pdfDoc: PDFDocument,
  page: PDFPage,
  url: string,
  rect: [number, number, number, number]
) {
  const context = page.doc.context;
  const targetPage = pdfDoc.getPage(parseInt(url) - 1);
  const targetPageRef = targetPage.ref;
  const destArray = context.obj([targetPageRef, PDFName.of('XYZ'), 0,
    ↪ 0, null]);

  const gotoAction = PDFDict.withContext(context);
  gotoAction.set(PDFName.of('S'), PDFName.of('GoTo'));
  gotoAction.set(PDFName.of('D'), destArray);

  // Create rectangle array for annotation
  const rectArray = PDFArray.withContext(context);
  rect.forEach((n) => rectArray.push(PDFNumber.of(n)));

  // Create annotation dictionary
  const linkAnnotation = PDFDict.withContext(context);
  linkAnnotation.set(PDFName.of('Type'), PDFName.of('Annot'));
  linkAnnotation.set(PDFName.of('Subtype'), PDFName.of('Link'));
  linkAnnotation.set(PDFName.of('Rect'), rectArray);
  linkAnnotation.set(PDFName.of('Border'), context.obj([0, 0, 0])); //
    ↪  No border
  linkAnnotation.set(PDFName.of('A'), gotoAction);

  // Add annotation dictionary to PDF and get reference
  const linkRef = context.register(linkAnnotation);

  // Add annotation to page
  const annotsRef = page.node.Annots();
  if (annotsRef) {
    const annotsArray: PDFArray = context.lookup(annotsRef, PDFArray);
    if (annotsArray) {
      annotsArray.push(linkRef);
    }
  } else {
    page.node.set(PDFName.of('Annots'), context.obj([linkRef]));
  }
}
```

6. Technological Details

6.1. Backend

- Language: Go (Golang) for performant concurrent backend services.
- Web Server: Built-in HTTP server with Gorilla WebSocket for real-time communication.

- Cloud: Firebase Firestore for NoSQL DB, Firebase Authentication for identity.
- Concurrency: Go channels and mutexes for shared state.

6.2. Frontend

- Framework: Svelte for reactive UI.
- Language: TypeScript and JavaScript.
- Communication: WebSocket integrated client.

6.3. Tooling and Build

Node.js-based toolchain (npm, pnpm) for dependencies and builds, environment-based backend Firebase config.

7. Testing

Automatic testing

The core backend components were extensively unit-tested to ensure correctness and robustness.

- **Canvas Service (`canvas_service.go`):**

  Unit tests verify all key behaviors such as creating a new service instance, adding and updating canvases, retrieving specific canvases, removing canvases, updating elements and backgrounds, and handling special actions on canvas elements. Tests cover both successful operations and failure cases (e.g., updating non-existent canvases). This ensures that the in-memory management of canvas state works flawlessly under concurrent access.

  Rationale: Confirm the integrity and behavior of canvas state management which is critical for real-time collaboration.

  Automation: Implemented with Go's testing framework (`testing` package).

  Requirements Tested: Functional requirements related to canvas creation, update, and state synchronization.

- **Models (User, Project, Invitation):**

  Tests validate parsing of HTTP requests into model structs, ensuring that JSON decoding handles normal and erroneous inputs correctly, including empty bodies or invalid JSON. They also verify the correctness of generated fields (like project IDs and invitation links) and edge cases for zero-value.

  Rationale: Prevent malformed data or format errors from impacting application stability.

  Requirements Tested: Functional correctness for request handling and basic validation.

Acceptance test

Before integrating the backend API with the frontend application, extensive testing was performed using **Insomnia**, a lightweight and user-friendly API client.

8. Deployment & User Guide

Installation and usage instructions are available in the repositories:

- Frontend: `https://github.com/drptms/phaint/blob/master/README.md`
- Backend: `https://github.com/drptms/phaint_be/blob/master/README.md`

9. Self-Evaluation

9.1. Farinelli Ettore

My primary contributions focused on developing the core collaborative drawing infrastructure and real-time synchronization systems that enable seamless multi-user interaction:

- **Drawing Interface and Tools Implementation**: Developed the complete drawing interface for the application, including all drawing tools and user interaction systems. This encompasses the frontend drawing canvas, tool selection mechanisms, and user input handling for various drawing operations.
- **Vector Graphics System**: Implemented the entire vector saving and usage system, where drawing elements are represented as lightweight metadata containing only essential geometric information (points, dimensions, and properties) rather than pixel data. This vector-based approach ensures scalability, editability, and efficient network transmission during collaborative sessions.
- **Hub Management System**: Architected and implemented the hub-based collaboration system where each project creates a dedicated hub instance. The system intelligently handles hub lifecycle - initializing new hubs with existing database content or connecting clients to active hubs and synchronizing current state to newly joined clients. This ensures consistent state management across all collaborative sessions.
- **Lightweight Communication Protocol**: Designed and implemented the client-server communication system using granular operations. Clients send minimal sub-operations (background changes, individual shape drawings, property modifications) rather than full canvas updates, optimizing network performance and ensuring rapid real-time collaboration with minimal latency.
- **Persistent State Management**: Implemented the automatic project persistence system that saves hub canvas state to the database every 4 seconds. This approach ensures data consistency and prevents loss while maintaining performance. The system follows a centralized update pattern where all modifications flow through the server before being applied locally, guaranteeing synchronized state across all clients.

- **Real-Time Cursor Visualization**: Developed the remote cursor tracking system that broadcasts user cursor movements to all connected clients within the same hub, providing visual awareness of collaborator activity and enhancing the collaborative experience.
- **Invitation and Collaboration System**: Implemented the complete invitation workflow, including invitation creation, secure link generation, and acceptance mechanisms that allow users to join collaborative projects and begin real-time editing sessions.

9.2. Ghignatti Nicolò

The primary contributions involved developing and integrating key backend functionalities that form the foundation of the platform's collaborative capabilities with some features that boost the project functionalities:

- **Firebase Connection Setup**: Established and configured the connection between the backend services and Firebase, enabling secure and scalable data persistence for users, projects, invitations, and canvas data.
- **User & Project Management**: Implemented core user features including registration, authentication via Firebase Auth, and profile handling, along with project creation and management, ensuring strong identity management and secure access control.
- **API Development**: Designed and built RESTful API endpoints to support CRUD operations on users, projects, and invitations, facilitating reliable client-server communication and data exchange.
- **WebSocket Real-Time Collaboration**: Developed the WebSocket handler and hub managing client connections, real-time message broadcasting, and synchronization of collaborative canvas state across multiple users.
- **UI Aesthetic Improvements**: Enhanced the visual appearance and usability of the frontend by contributing aesthetic refinements, improving user experience during interaction with the collaborative drawing environment.
- **Object Enhancement**: Implemented the possibility to enhance the drawings in the canvas with useful actions.
- **PDF export**: Implemented functionality to export collaborative drawings, including enhancements, into downloadable and shareable PDF files.